Learn JavaScript

Published with GitBook

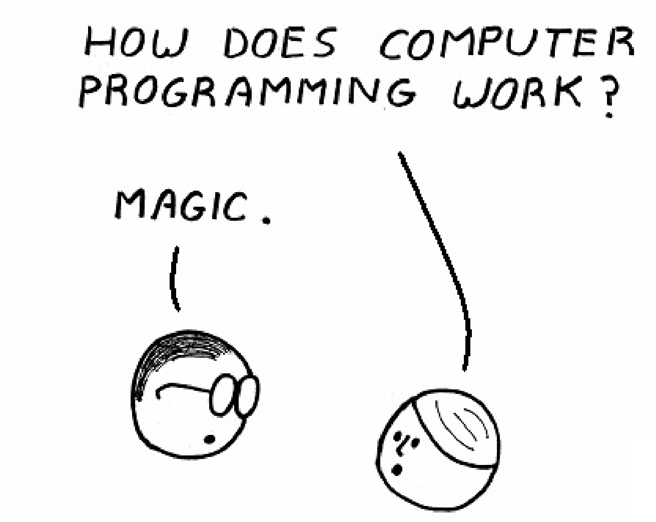
Table of Contents

Introduction	1.1
Basics	1.2
Comments	1.2.1
Variables	1.2.2
Types	1.2.3
Equality	1.2.4
Numbers	1.3
Creation	1.3.1
Basic Operators	1.3.2
Advanced Operators	1.3.3
Strings	1.4
Creation	1.4.1
Concatenation	1.4.2
Length	1.4.3
Conditional Logic	1.5
If	1.5.1
Else	1.5.2
Comparators	1.5.3
Concatenate	1.5.4
Arrays	1.6
Indices	1.6.1
Length	1.6.2
Loops	1.7
For	1.7.1
While	1.7.2
Do...While	1.7.3
Functions	1.8
Declare	1.8.1
Higher order	1.8.2
Objects	1.9

Creation	1.9.1
Properties	1.9.2
Mutable	1.9.3
Reference	1.9.4
Prototype	1.9.5
Delete	1.9.6
Enumeration	1.9.7
Global footprint	1.9.8

Learn Javascript

This book will teach you the basics of programming and Javascript. Whether you are an experienced programmer or not, this book is intended for everyone who wishes to learn the JavaScript programming language.



JavaScript (*JS for short*) is the programming language that enables web pages to respond to user interaction beyond the basic level. It was created in 1995, and is today one of the most famous and used programming languages.

Basics about Programming

In this first chapter, we'll learn the basics of programming and the Javascript language.

Programming means writing code. A book is made up of chapters, paragraphs, sentences, phrases, words and finally punctuation and letters, likewise a program can be broken down into smaller and smaller components. For now, the most important is a statement. A statement is analogous to a sentence in a book. On its own, it has structure and purpose, but without the context of the other statements around it, it isn't that meaningful.

A statement is more casually (and commonly) known as a *line of code*. That's because statements tend to be written on individual lines. As such, programs are read from top to bottom, left to right. You might be wondering what code (also called source code) is. That happens to be a broad term which can refer to the whole of the program or the smallest part. Therefore, a line of code is simply a line of your program.

Here is a simple example:

```
var hello = "Hello";
var world = "World";

// Message equals "Hello World"
var message = hello + " " + world;
```

This code can be executed by another program called an *interpreter* that will read the code, and execute all the statements in the right order.

Comments

Comments are statements that will not be executed by the interpreter, comments are used to mark annotations for other programmers or small descriptions of what your code does, thus making it easier for others to understand what your code does.

In Javascript, comments can be written in 2 different ways:

- Line starting with `//` :

```
// This is a comment, it will be ignored by the interpreter
var a = "this is a variable defined in a statement";
```

- Section of code starting with `/*` and ending with `*/` , this method is used for multi-line comments:

```
/*
This is a multi-line comment,
it will be ignored by the interpreter
*/
var a = "this is a variable defined in a statement";
```

Exercise

Mark the editor's contents as a comment

```
Mark me as a comment
or I'll throw an error
```

Variables

The first step towards really understanding programming is looking back at algebra. If you remember it from school, algebra starts with writing terms such as the following.

```
3 + 5 = 8
```

You start performing calculations when you introduce an unknown, for example x below:

```
3 + x = 8
```

Shifting those around you can determine x:

```
x = 8 - 3
-> x = 5
```

When you introduce more than one you make your terms more flexible - you are using variables:

```
x + y = 8
```

You can change the values of x and y and the formula can still be true:

```
x = 4
y = 4
```

or

```
x = 3
y = 5
```

The same is true for programming languages. In programming, variables are containers for values that change. Variables can hold all kind of values and also the results of computations. Variables have a name and a value separated by an equals sign (=). Variable names can be any letter or word, but bear in mind that there are restrictions from language to language of what you can use, as some words are reserved for other functionality.

Let's check out how it works in Javascript, The following code defines two variables, computes the result of adding the two and defines this result as a value of a third variable.

```
var x = 5;
var y = 6;
var result = x + y;
```

Variable types

Computers are sophisticated and can make use of more complex variables than just numbers. This is where variable types come in. Variables come in several types and different languages support different types.

The most common types are:

- **Numbers**
  - **Float**: a number, like 1.21323, 4, -33.5, 100004 or 0.123
  - **Integer**: a number like 1, 12, -33, 140 but not 1.233
- **String**: a line of text like "boat", "elephant" or "damn, you are tall!"
- **Boolean**: either true or false, but nothing else
- **Arrays**: a collection of values like: 1,2,3,4,'I am bored now'
- **Objects**: a representation of a more complex object
- **null**: a variable that contains null contains no valid Number, String, Boolean, Array, or Object
- **undefined**: the undefined value is obtained when you use an object property that does not exist, or a variable that has been declared, but has no value assigned to it.

JavaScript is a *"loosely typed"* language, which means that you don't have to explicitly declare what type of data the variables are. You just need to use the `var` keyword to indicate that you are declaring a variable, and the interpreter will work out what data type you are using from the context, and use of quotes.

Exercise

Create a variable named \`a\` using the keyword \`var\`.

```

```

Equality

Programmers frequently need to determine the equality of variables in relation to other variables. This is done using an equality operator.

The most basic equality operator is the `==` operator. This operator does everything it can to determine if two variables are equal, even if they are not of the same type.

For example, assume:

```
var foo = 42;
var bar = 42;
var baz = "42";
var qux = "life";
```

`foo == bar` will evaluate to `true` and `baz == qux` will evaluate to `false`, as one would expect. However, `foo == baz` will *also* evaluate to `true` despite `foo` and `baz` being different types. Behind the scenes the `==` equality operator attempts to force its operands to the same type before determining their equality. This is in contrast to the `===` equality operator.

The `===` equality operator determines that two variables are equal if they are of the same type *and* have the same value. With the same assumptions as before, this means that `foo === bar` will still evaluate to `true`, but `foo === baz` will now evaluate to `false`. `baz === qux` will still evaluate to `false`.

Numbers

JavaScript has **only one type of numbers** – 64-bit float point. It's the same as Java's `double`. Unlike most other programming languages, there is no separate integer type, so 1 and 1.0 are the same value.

In this chapter, we'll learn how to create numbers and perform operations on them (like additions and subtractions).

Creation

Creating a number is easy, it can be done just like for any other variable type using the `var` keyword.

Numbers can be created from a constant value:

```
// This is a float:
var a = 1.2;

// This is an integer:
var b = 10;
```

Or from the value of another variable:

```
var a = 2;
var b = a;
```

Exercise

Create a variable `x` which equals `10` and create a variable `y` which equals `a`.

```
var a = 11;
```

Operators

You can apply mathematic operations to numbers using some basic operators like:

- **Addition**: `c = a + b`
- **Subtraction**: `c = a - b`
- **Multiplication**: `c = a * b`
- **Division**: `c = a / b`

You can use parentheses just like in math to separate and group expressions: `c = (a / b) + d`

Exercise

Create a variable `x` equal to the sum of `a` and `b` divided by `c` and finally multiplied by `d`.

```
var a = 2034547;
var b = 1.567;
var c = 6758.768;
var d = 45084;

var x =
```

Advanced Operators

Some advanced operators can be used, such as:

- **Modulus (division remainder)**: `x = y % 2`
- **Increment**: Given a = 5
  - `c = a++` , Results: c = 5 and a = 6
  - `c = ++a` , Results: c = 6 and a = 6
- **Decrement**: Given a = 5
  - `c = a--` , Results: c = 5 and a = 4
  - `c = --a` , Results: c = 4 and a = 4

Exercise

Define a variable `c` as the modulus of the decremented value of `x` by 3.

```
var x = 10;

var c =
```

Strings

JavaScript strings share many similarities with string implementations from other high-level languages. They represent text based messages and data.

In this course we will cover the basics. How to create new strings and perform common operations on them.

Here is an example of a string:

```
"Hello World"
```

Creation

You can define strings in JavaScript by enclosing the text in single quotes or double quotes:

```
// Single quotes can be used
var str = 'Our lovely string';

// Double quotes as well
var otherStr = "Another nice string";
```

In Javascript, Strings can contain UTF-8 characters:

```
"中文 español English हिन्दी العربية português বাংলা русский 日本語 ▫▫▫▫▫▫ 한국어";
```

Note: Strings can not be subtracted, multiplied or divided.

Exercise

Create a variable named `str` set to the value `"abc"`.

Concatenation

Concatenation involves adding two or more strings together, creating a larger string containing the combined data of those original strings. This is done in JavaScript using the **+** operator.

```
var bigStr = 'Hi ' + 'JS strings are nice ' + 'and ' + 'easy to add';
```

Exercise

Add up the different names so that the `fullName` variable contains John's complete name.

```
var firstName = "John";
var lastName = "Smith";

var fullName =
```

Length

It's easy in Javascript to know how many characters are in string using the property `.length`.

```
// Just use the property .length
var size = 'Our lovely string'.length;
```

Note: Strings can not be substracted, multiplied or divided.

Exercise

Store in the variable named `size` the length of `str`.

```
var str = "Hello World";

var size =
```

Conditional Logic

A condition is a test for something. Conditions are very important for programming, in several ways:

First of all conditions can be used to ensure that your program works, regardless of what data you throw at it for processing. If you blindly trust data, you’ll get into trouble and your programs will fail. If you test that the thing you want to do is possible and has all the required information in the right format, that won’t happen, and your program will be a lot more stable. Taking such precautions is also known as programming defensively.

The other thing conditions can do for you is allow for branching. You might have encountered branching diagrams before, for example when filling out a form. Basically, this refers to executing different “branches” (parts) of code, depending on if the condition is met or not.

In this chapter, we'll learn the base of conditional logic in Javascript.

Condition If

The easiest condition is an if statement and its syntax is `if(condition){ do this … }`. The condition has to be true for the code inside the curly braces to be executed. You can for example test a string and set the value of another string dependent on its value:

```
var country = 'France';
var weather;
var food;
var currency;

if(country === 'England') {
    weather = 'horrible';
    food = 'filling';
    currency = 'pound sterling';
}

if(country === 'France') {
    weather = 'nice';
    food = 'stunning, but hardly ever vegetarian';
    currency = 'funny, small and colourful';
}

if(country === 'Germany') {
    weather = 'average';
    food = 'wurst thing ever';
    currency = 'funny, small and colourful';
}

var message = 'this is ' + country + ', the weather is ' +
            weather + ', the food is ' + food + ' and the ' +
            'currency is ' + currency;
```

Note: Conditions can also be nested.

Exercise

Fill up the value of `name` to validate the condition.

```
var name =

if (name === "John") {

}
```

Else

There is also an `else` clause that will be applied when the first condition isn’t true. This is very powerful if you want to react to any value, but single out one in particular for special treatment:

```
var umbrellaMandatory;

if(country === 'England'){
    umbrellaMandatory = true;
} else {
    umbrellaMandatory = false;
}
```

The `else` clause can be joined with another `if`. Lets remake the example from the previous article:

```
if(country === 'England') {
    ...
} else if(country === 'France') {
    ...
} else if(country === 'Germany') {
    ...
}
```

Exercise

Fill up the value of \`name\` to validate the \`else\` condition.

```
var name =

if (name === "John") {

} else if (name === "Aaron") {
    // Valid this condition
}
```

Comparators

Lets now focus on the conditional part:

```
if (country === "France") {
    ...
}
```

The conditional part is the variable `country` followed by the three equal signs (`===`). Three equal signs tests if the variable `country` has both the correct value (`France`) and also the correct type (`String`). You can test conditions with double equal signs, too, however a conditional such as `if (x == 5)` would then return true for both `var x = 5;` and `var x = "5";` . Depending on what your program is doing, this could make quite a difference. It is highly recommended as a best practice that you always compare equality with three equal signs (`===` and `!==`) instead of two (`==` and `!=`).

Other conditional test:

- `x > a` : is x bigger than a?
- `x < a` : is x less than a?
- `x <= a` : is x less than or equal to a?
- `x >=a` : is x greater than or equal to a?
- `x != a` : is x not a?
- `x` : does x exist?

Exercise

Add a condition to change the value of `a` to the number 10 if `x` is bigger than 5.

```
var x = 6;
var a = 0;
```

Logical Comparison

In order to avoid the if-else hassle, simple logical comparisons can be utilised.

```
var topper = (marks > 85) ? "YES" : "NO";
```

In the above example, `?` is a logical operator. The code says that if the value of marks is greater than 85 i.e. `marks > 85` , then `topper = YES` ; otherwise `topper = NO` . Basically, if the comparison condition proves true, the first argument is accessed and if the comparison condition is false , the second argument is accessed.

Concatenate conditions

Furthermore you can concatenate different conditions with "or” or “and” statements, to test whether either statement is true, or both are true, respectively.

In JavaScript “or” is written as `||` and “and” is written as `&&`.

Say you want to test if the value of x is between 10 and 20—you could do that with a condition stating:

```
if(x > 10 && x < 20) {
    ...
}
```

If you want to make sure that country is either “England” or “Germany” you use:

```
if(country === 'England' || country === 'Germany') {
    ...
}
```

Note: Just like operations on numbers, Condtions can be grouped using parenthesis, ex: `if ( (name === "John" || name === "Jennifer") && country === "France")`.

Exercise

Fill up the 2 conditions so that `primaryCategory` equals `"E/J"` only if name equals `"John"` and country is `"England"`, and so that `secondaryCategory` equals `"E|J"` only if name equals `"John"` or country is `"England"`

```
var name = "John";
var country = "England";
var primaryCategory, secondaryCategory;

if ( /* Fill here */ ) {
    primaryCategory = "E/J";
}
if ( /* Fill here */ ) {
    secondaryCategory = "E|J";
}
```

Arrays

Arrays are a fundamental part of programming. An array is a list of data. We can store a lot of data in one variable, which makes our code more readable and easier to understand. It also makes it much easier to perform functions on related data.

The data in arrays are called **elements**.

Here is a simple array:

```
// 1, 1, 2, 3, 5, and 8 are the elements in this array
var numbers = [1, 1, 2, 3, 5, 8];
```

Indices

So you have your array of data elements, but what if you want to access a specific element? That is where indices come in. An **index** refers to a spot in the array. indices logically progress one by one, but it should be noted that the first index in an array is 0, as it is in most languages. Brackets [] are used to signify you are referring to an index of an array.

```
// This is an array of strings
var fruits = ["apple", "banana", "pineapple", "strawberry"];

// We set the variable banana to the value of the second element of
// the fruits array. Remember that indices start at 0, so 1 is the
// second element. Result: banana = "banana"
var banana = fruits[1];
```

Exercise

Define the variables using the indices of the array

```
var cars = ["Mazda", "Honda", "Chevy", "Ford"]
var honda =
var ford =
var chevy =
var mazda =
```

Length

Arrays have a property called length, and it's pretty much exactly as it sounds, it's the length of the array.

```
var array = [1 , 2, 3];

// Result: l = 3
var l = array.length;
```

Exercise

Define the variable a to be the number value of the length of the array

```
var array = [1, 1, 2, 3, 5, 8];
var l = array.length;
var a =
```

Loops

Loops are repetitive conditions where one variable in the loop changes. Loops are handy, if you want to run the same code over and over again, each time with a different value.

Instead of writing:

```
doThing(cars[0]);
doThing(cars[1]);
doThing(cars[2]);
doThing(cars[3]);
doThing(cars[4]);
```

You can write:

```
for (var i=0; i < cars.length; i++) {
    doThing(cars[i]);
}
```

For Loop

The easiest form of a loop is the for statement. This one has a syntax that is similar to an if statement, but with more options:

```
for(condition; end condition; change){
    // do it, do it now
}
```

Lets for example see how to execute the same code ten-times using a `for` loop:

```
for(var i = 0; i < 10; i = i + 1){
    // do this code ten-times
}
```

> ***Note***: `i = i + 1` can be written `i++`.

Exercise

Using a for-loop, create a variable named `message` that equals the concatenation of integers (0, 1, 2, ...) from 0 to 99.

```
var message = "";
```

While Loop

While Loops repetitively execute a block of code as long as a specified condition is true.

```
while(condition){
    // do it as long as condition is true
}
```

For example, the loop in this example will repetitively execute its block of code as long as the variable i is less than 5:

```
var i = 0, x = "";
while (i < 5) {
    x = x + "The number is " + i;
    i++;
}
```

The Do/While Loop is a variant of the while loop. This loop will execute the code block once before checking if the condition is true. It then repeats the loop as long as the condition is true:

```
do {
    // code block to be executed
} while (condition);
```

Note: Be careful to avoid infinite looping if the condition is always true!

Exercise

Using a while-loop, create a variable named `message` that equals the concatenation of integers (0, 1, 2, ...) as long as its length (`message.length`) is less than 100.

```
var message = "";
```

Do...While Loop

The do...while statement creates a loop that executes a specified statement until the test condition evaluates to be false. The condition is evaluated after executing the statement. Syntax for do... while is

```
do{
    // statement
}
while(expression) ;
```

Lets for example see how to print numbers less than 10 using `do...while` loop:

```
var i = 0;
do {
    document.write(i + " ");
    i++; // incrementing i by 1
} while (i < 10);
```

> ***Note***: `i = i + 1` can be written `i++`.

Exercise

Using a do...while-loop, print numbers between less than 5.

```
var i = 0;
```

Functions

Functions, are one of the most powerful and essential notions in programming.

Functions like mathematical functions perform transformations, they take input values called **arguments** and **return** an output value.

Declaring Functions

Functions, like variables, must be declared. Let's declare a function `double` that accepts an **argument** called `x` and **returns** the double of x :

```
function double(x) {
    return 2 * x;
}
```

> *Note:* the function above **may** be referenced before it has been defined.

Functions are also values in JavaScript; they can be stored in variables (just like numbers, strings, etc ...) and given to other functions as arguments :

```
var double = function(x) {
    return 2 * x;
};
```

> *Note:* the function above **may not** be referenced before it is defined, just like any other variable.

Exercise

Declare a function named `triple` that takes an argument and returns its triple.

Higher Order Functions

Higher order functions are functions that manipulate other functions. For example, a function can take other functions as arguments and/or produce a function as its return value. Such *fancy* functional techniques are powerful constructs available to you in JavaScript and other high-level languages like python, lisp, etc.

We will now create two simple functions, `add_2` and `double`, and a higher order function called `map`. `map` will accept two arguments, `func` and `list` (its declaration will therefore begin `map(func,list)`), and return an array. `func` (the first argument) will be a function that will be applied to each of the elements in the array `list` (the second argument).

```
// Define two simple functions
var add_2 = function(x) {
    return x + 2;
};
var double = function(x) {
    return 2 * x;
};

// map is cool function that accepts 2 arguments:
//  func    the function to call
//  list    a array of values to call func on
var map = function(func, list) {
    var output=[];              // output list
    for(idx in list) {
        output.push( func(list[idx]) );
    }
    return output;
}


// We use map to apply a function to an entire list
// of inputs to "map" them to a list of corresponding outputs
map(add_2, [5,6,7]) // => [7, 8, 9]
map(double, [5,6,7]) // => [10, 12, 14]
```

The functions in the above example are simple. However, when passed as arguments to other functions, they can be composed in unforeseen ways to build more complex functions.

For example, if we notice that we use the invocations `map(add_2, ...)` and `map(double, ...)` very often in our code, we could decide we want to create two special-purpose list processing functions that have the desired operation baked into them. Using function composition, we could do this as follows:

```
process_add_2 = function(list) {
    return map(add_2, list);
}
process_double = function(list) {
    return map(double, list);
}
process_add_2([5,6,7]) // => [7, 8, 9]
process_double([5,6,7]) // => [10, 12, 14]
```

Now let's create a function called `buildProcessor` that takes a function `func` as input and returns a `func` -processor, that is, a function that applies `func` to each input in list.

```
// a function that generates a list processor that performs
var buildProcessor = function(func) {
    var process_func = function(list) {
        return map(func, list);
    }
    return process_func;
}
// calling buildProcessor returns a function which is called with a list input


// using buildProcessor we could generate the add_2 and double list processors as foll
ows:
process_add_2 = buildProcessor(add_2);
process_double = buildProcessor(double);

process_add_2([5,6,7]) // => [7, 8, 9]
process_double([5,6,7]) // => [10, 12, 14]
```

Let's look at another example. We'll create a function called `buildMultiplier` that takes a number `x` as input and returns a function that multiplies its argument by `x` :

```
var buildMultiplier = function(x) {
    return function(y) {
        return x * y;
    }
}

var double = buildMultiplier(2);
var triple = buildMultiplier(3);

double(3); // => 6
triple(3); // => 9
```

Exercise

> Define a function named `negate` that takes `add1` as argument and returns a function, that returns the negation of the value returned by `add1`. (Things get a bit more complicated ;))

```
var add1 = function (x) {
    return x + 1;
};

var negate = function(func) {
    // TODO
};

// Should return -6
// Because (5+1) * -1 = -6
negate(add1)(5);
```

Objects

The primitive types of JavaScript are `true`, `false`, numbers, strings, `null` and `undefined`. **Every other value is an `object`.**

In JavaScript objects contain `propertyName`: `propertyValue` pairs.

Creation

There are two ways to create an `object` in JavaScript:

1. literal

    ```
    var object = {};
      // Yes, simply a pair of curly braces!
    ```

    > ***Note:*** this is the **recomended** way.

2. and object-oriented

    ```
    var object = new Object();
    ```

    > ***Note:*** it's almost like Java.

Properties

Object's property is a `propertyName` : `propertyValue` pair, where **property name can be only a string**. If it's not a string, it gets casted into a string. You can specify properties **when creating** an object **or later**. There may be zero or more properties separated by commas.

```
var language = {
    name: 'JavaScript',
    isSupportedByBrowsers: true,
    createdIn: 1995,
    author:{
        firstName: 'Brendan',
        lastName: 'Eich'
    },
 // Yes, objects can be nested!
    getAuthorFullName: function(){
        return this.author.firstName + " " + this.author.lastName;
    }
 // Yes, functions can be values too!
};
```

The following code demonstates how to **get** a property's value.

```
var variable = language.name;
 // variable now contains "JavaScript" string.
    variable = language['name'];
 // The lines above do the same thing. The difference is that the second one lets you
use litteraly any string as a property name, but it's less readable.
    variable = language.newProperty;
 // variable is now undefined, because we have not assigned this property yet.
```

The following example shows how to **add** a new property **or change** an existing one.

```
language.newProperty = 'new value';
 // Now the object has a new property. If the property already exists, its value will
be replaced.
language['newProperty'] = 'changed value';
 // Once again, you can access properties both ways. The first one (dot notation) is r
ecomended.
```

Mutable

The difference between objects and primitive values is that **we can change objects**, whereas primitive values are immutable.

```
var myPrimitive = "first value";
    myPrimitive = "another value";
 // myPrimitive now points to another string.
var myObject = { key: "first value"};
    myObject.key = "another value";
 // myObject points to the same object.
```

Reference

Objects are **never copied**. They are passed around by reference.

```
// Imagine I had a pizza
var myPizza = {slices: 5};
 // And I shared it with You
var yourPizza = myPizza;
 // I eat another slice
myPizza.slices = myPizza.slices - 1;
var numberOfSlicesLeft = yourPizza.slices;
 // Now We have 4 slices because myPizza and yourPizza
 // reference to the same pizza object.
var a = {}, b = {}, c = {};
 // a, b, and c each refer to a
 // different empty object
a = b = c = {};
 // a, b, and c all refer to
 // the same empty object
```

Prototype

Every object is linked to a prototype object from which it inherits properties.

All objects created from object literals (`{}`) are automatically linked to Object.prototype, which is an object that comes standard with JavaScript.

When a JavaScript interpreter (a module in your browser) tries to find a property, which You want to retrieve, like in the following code:

```
var adult = {age: 26},
    retrievedProperty = adult.age;
 // The line above
```

First, the interpreter looks through every property the object itself has. For example, `adult` has only one own property — `age` . But besides that one it actually has a few more properties, which were inherited from Object.prototype.

```
var stringRepresentation = adult.toString();
 // the variable has value of '[object Object]'
```

`toString` is an Object.prototype's property, which was inherited. It has a value of a function, which returns a string representation of the object. If you want it to return a more meaningful representation, then you can override it. Simply add a new property to the adult object.

```
adult.toString = function(){
    return "I'm "+this.age;
}
```

If you call the `toString` function now, the interpreter will find the new property in the object itself and stop.

Thus the interpreter retrieves the first property it will find on the way from the object itself and further through its prototype.

To set your own object as a prototype instead of the default Object.prototype, you can invoke `Object.create` as follows:

```
var child = Object.create(adult);
 /* This way of creating objects lets us easily replace the default Object.prototype w
ith the one we want. In this case, the child's prototype is the adult object. */
child.age = 8;
 /* Previously, child didn't have its own age property, and the interpreter had to loo
k further to the child's prototype to find it.
 Now, when we set the child's own age, the interpreter will not go further.
 Note: adult's age is still 26. */
var stringRepresentation = child.toString();
 // The value is "I'm 8".
 /* Note: we have not overridden the child's toString property, thus the adult's metho
d will be invoked. If adult did not have toString property, then Object.prototype's to
String method would be invoked, and we would get "[object Object]" instead of "I'm 8"
*/
```

`child`'s prototype is `adult`, whose prototype is `Object.prototype`. This sequence of prototypes is called **prototype chain**.

Delete

`delete` can be used to **remove a property** from an object. It will remove a property from the object if it has one. It will not look further in the prototype chain. Removing a property from an object may allow a property from the prototype chain to shine through:

```
var adult = {age:26},
    child = Object.create(adult);
    child.age = 8;

delete child.age;
 /* Remove age property from child, revealing the age of the prototype, because then i
t is not overriden. */
var prototypeAge = child.age;
 // 26, because child does not have its own age property.
```

Enumeration

The `for in` statement can loop over all of the property names in an object. The enumeration will include functions and prototype properties.

```
var fruit = {
    apple: 2,
    orange:5,
    pear:1
},
sentence = 'I have ',
quantity;
for (kind in fruit){
    quantity = fruit[kind];
    sentence += quantity+' '+kind+
                (quantity===1?'':'s')+
                ', ';
}
 // The following line removes the trailing coma.
sentence = sentence.substr(0,sentence.length-2)+'.';
 // I have 2 apples, 5 oranges, 1 pear.
```

Global footprint

If you are developing a module, which might be running on a web page, which also runs other modules, then you must beware the variable name overlapping.

Suppose we are developing a counter module:

```
var myCounter = {
    number : 0,
    plusPlus : function(){
        this.number : this.number + 1;
    },
    isGreaterThanTen : function(){
        return this.number > 10;
    }
}
```

> ***Note:*** this technique is often used with closures, to make the internal state immutable from the outside.

The module now takes only one variable name — `myCounter`. If any other module on the page makes use of such names like `number` or `isGreaterThanTen` then it's perfectly safe, because we will not override each others values;